Appears in *Proceedings, 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)*, September 2006, Chicago, IL, USA.

Model-Driven Agile Development of Reactive Multi-Agent Systems

James Kirby, Jr.
Naval Research Laboratory
james.kirby@nrl.navy.mil

Abstract

The Sage development method and associated tool set support an incremental, iterative, model-driven process to build and maintain high assurance, reactive multi-agent systems. A set of interconnected models provide documentation supporting high assurance certification efforts, maintenance, and reuse. Tools can analyze the models for important classes of errors, and generate complete multi-agent systems.

1. Introduction

We are concerned with developing high assurance, reactive multi-agent systems (MAS). Agents comprising such systems monitor their environment via hardware and software sensors, and react to it via hardware and software actuators. An agent may maintain state and send values to and receive values from other agents. Individual agents, or the entire MAS, may exhibit complex modal behavior.

For high assurance systems, just delivering a working system is insufficient. The customer must have confidence that it has certain critical properties (e.g, security, safety). Some of that confidence is derived from documentation that is typically a byproduct of the development process, e.g., design specifications, test plans. Assurance arguments use as evidence the documentation and code, and the results of applying automated analysis tools (e.g., theorem provers, model checkers) to the documentation and code.

Agile development [2][23] is a process focused on frequent delivery of working software and on responsiveness to customer needs. Frequent delivery of working software helps distinguish progress from motion, which can be difficult to distinguish in a document-centered process that produces working code only near the end of development. Agile development is code-centric; it largely forgoes developing and delivering work products other than code, e.g., requirements and design models.

Model-driven development is a software development approach that is agnostic with respect to process and methods. Its "defining characteristic is that software development's primary focus and products are models rather than computer programs." [25].

As with software development in general, high assurance software development suffers from the sort of problems that agile development's ability to distinguish progress and motion can ameliorate. High assurance projects go over budget and schedule, fail to deliver promised functionality on time, or fail to deliver at all. Being able to discern early that while the project is generating lots of documents, it is making insufficient progress toward delivering a system can make it possible to fix project problems before it's too late. However, applying agile development to high assurance software is problematic: a key principle eschews exactly the documentation that assurance arguments require as evidence.

Sage marries agile development to model-driven development. The Sage meta-model, developed to facilitate model-driven development, provides a template for recording decisions developers need to make, which supports faking a rational development process [20]. Further, the meta-model, by eliminating redundant capture of developers' decisions, supports agility. Executable code, generated from developers' descriptions of software behavior, can be developed, delivered, and updated quickly. Organizing the behavior into, e.g., design elements supporting intellectual control required of assurance arguments, can be done concurrently or subsequently. Developers practice agile development, albeit with models rather than a traditional programming language, while producing and maintaining documentation required by assurance arguments. Generating executable code from the models gives confidence that the models are consistent with the executable code. Automated analyses of such models can give confidence that analysis results are relevant to the executable code.

This paper is about the Sage development method and its associated tool set, which support an incremental, iterative, model-driven process to build and maintain high assurance multi-agent systems. A Sage *application model* comprises a set of interconnected models that provide documentation supporting high assurance certification efforts, that tools can analyze for important classes of errors, that support maintenance and reuse of the models, and from which tools can generate a complete MAS.

2. The Sage Models

A Sage application model, an instantiation of the Sage meta-model [3], records developers' decisions in four distinct models. The *environmental model* records the boundary of the system with its environment. The *behavioral model* records the behavior (i.e., functionality, business logic) of the system. The *design model* records the decomposition of the behavior into pieces supporting intellectual control and encapsulation. The *run-time model* records the decomposition of the behavior into agents supporting performance and quality of service requirements.

In various combinations, the four models share subsets of a set of mathematical variables called *attributes*. Some of the attributes denote quantities and qualities in the environment (e.g., the position of a switch, the brightness of an office), some denote physical inputs and outputs, others represent quantities and qualities chosen for the convenience of modeling. Mathematical functions that specify the values of selected attributes model the behavior of the system and its components. Automation can reflect changes to an attribute declaration or function in one model to all models that share the attribute. Changes to other aspects of a model, which have to do with concerns unique to that model, do not affect other models.

Sage adopts a small subset of UML—class diagrams—to provide views of three of the four Sage models. Associated with each class representing a design element in a Sage model may be a number of attributes. In a Sage design or run-time model the attributes are organized into named compartments. Two of the compartments, *provides output* and *provides input*, provide an *abstract interface* [7] for the design element the class represents. The *provides output* compartment lists attributes whose values the design element makes available to other design elements in the model. The *provides input* compartment lists attributes whose values other design elements may set. Two more compartments, *requires input* and *requires output*, describe what the design element needs to satisfy its abstract interface. The *requires input* compartment lists attributes whose values the design element requires. The design element is responsible for calculating the values of attributes listed in the *requires output* compartment. It depends upon others to set them. A fifth compartment, *local*, lists attributes whose values are computed and used only by the design element containing the compartment.

Figure 1 illustrates a UML class representing an agent taken from a Sage run-time model. The class *FMControlPanel* has four named compartments, *provides output*, *requires input*, *requires output*, and *local*. The third compartment, *requires output*, contains the attributes *oMDMalfunction* and *oOLSMalfunction*. *FMControlPanel* is responsible for calculating the values of these attributes.

2.1. The Environmental Model

The environmental model provides an application-specific ontology for the MAS, including descriptions of what the MAS can sense, control, or affect. The model records the system boundary by identifying objects in the environment of the system (which may include the system itself and components of the system) and attributes of the objects that may be relevant to the system, referred to as *environmental attributes*. The declaration of an attribute in the model includes its type, which characterizes the values it can assume, and a description of how to interpret its value [15].

UML classes represent the objects in the system environment. The attributes associated with each object are listed in the corresponding class. Standard UML class notation may record relationships among the classes of the environmental model, the relative cardinality of the objects abstracted by the classes of the environmental model, and the cardinality of the attributes of each object.

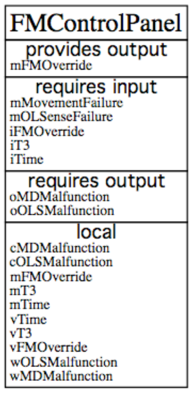

FIGURE 1. Class with named compartments

2.2. The Behavioral Model

The behavioral model describes the behavior (i.e., the functionality or business logic) of the MAS as a whole. The model includes declarations of attributes, types, and constants and functions which specify the values of attributes and the effects of setting their values. References [13] and [14] provide details on how to represent such functions and on their formal semantics.

Sage models behavior of an MAS in terms of selected environmental attributes. Sage records which environmental attributes the system is to control, manipulate, or affect. These attributes are referred to as *controlled attributes*. The values the controlled attributes assume over time is the system behavior. Sage records which environmental attributes determine the values of the controlled attributes. These attributes are referred to as *monitored attributes*.

The behavioral model includes declarations of additional attributes, called *terms*, assigning them names and types that characterize the values that they can assume. For a mode-dependent system, one whose behavior varies significantly depending upon the system's mode of operation (e.g., initialization, normal operation, alarm), one or more attributes called *mode classes* may capture the modes. The declaration of each mode class gives it a name and a type, which identifies the names of the modes in the mode class.

The domain of each function specifying the value of a controlled attribute may comprise monitored attributes, controlled attributes, mode classes, and terms (denoted M, C, Z, and T, respectively). Additional mathematical functions specify the values of the mode classes and the terms. The domain of each of these functions may comprise monitored attributes, controlled attributes, mode classes, and terms.

The behavioral model includes declarations of additional attributes called *virtual inputs*, *virtual outputs*, *physical inputs*, and *physical outputs* (denoted V, W, I, and O, respectively). The declaration of each of these additional attributes identifies its name and type. Physical inputs and outputs denote interfaces to physical devices, other systems, and other software. Virtual inputs and outputs provide stable virtual devices that abstract from aspects of the physical devices, other systems, and other software that is likely to change. Functions specify how to calculate the values of the monitored attributes, virtual outputs, and physical outputs.

2.3. The Design Model

The design model records a decomposition of the behavioral model into *design classes*—design elements responsible for distinct concerns. This decomposition of behavior is independent of its run-time organization. References [15][22][7] provide rationale and more detail for this decomposition into *information hiding modules*, which we represent with UML classes and aggregation. (The designer may choose other decomposition criteria.) As in [15], the design model is an aggregation of Function Driver, Mode Determination, System Value, and Device Interface classes. Any or all of the classes may be further decomposed into subclasses. The modules that [15] describes as "left out"—Extended Computer and Software Decision—Sage considers to be a responsibility of the middleware.

Designing a design class is recorded by assigning attributes to the five named compartments. Assigning attributes to the *provides output*, *requires output*, and *local* compartments of the design classes records the design model's decomposition of the behavioral model. Implicit in assigning an attribute to one of these compartments is the assigning of the corresponding value function to the design class.

Function Driver classes determine the values of the controlled attributes, which are the behavior of the MAS. Each controlled attribute is listed in the *requires output* compartment of a function driver class, which is responsible for calculating its value. Attributes required by the functions are listed in the *local* or *requires input* compartment. Each of the latter attributes must appear in the *provides output* compartment of some other class which calculates its value.

System Value classes are responsible for providing the values of attributes shared by several classes, and for setting the values of controlled attributes. The *provides output* compartment lists the shared attributes. The appropriate functions calculate their values. The *provides input* compartment lists the controlled attributes. When a Function Driver class sets the value of a controlled attribute, the system value class sets the value of one or more virtual outputs. The *requires input* compartment lists attributes whose values the various functions require that aren't calculated in the class. In particular, this will include virtual inputs (V). The *requires output* compartment lists the virtual outputs (W) whose values the class calculates. The *local* compartment lists attributes used within the class that aren't used by other classes.

The *provides output* compartment of the Mode Determination class lists mode class attributes (Z). Attributes required to calculate Z are listed in the *local* compartment if they are only required by one of the functions. Otherwise, they are listed in the *requires input* compartment.

Each virtual input (*V*) appears in the *provides output* compartment of one Device Interface class. The appropriate functions calculate the values of the virtual inputs in terms of the physical inputs (*I*). The *requires input* compartment of each Device Interface class lists physical inputs attributes (*I*). Providing the values of the physical inputs may be a responsibility of the middleware. Each virtual output, *W*, appears in the *provides input* compartment of a Device Interface class. The *requires output* compartment of each Device Interface class lists the appropriate members of *O*, the physical outputs. When a System Value class sets the value of virtual output, a Device Interface class sets the value of one or more physical outputs. It may be the responsibility of the middleware to send the values of the output attributes to the appropriate physical devices.

2.4. The Run-Time Model

The run-time model records the decomposition of the behavioral model—a network of functions, each of which calculates the value of one attribute in terms of the values of other attributes—into agents. Performance and quality of service goals guide the development of agents. Aside from agents that require special resources (e.g., amounts of memory available only on certain processors, communications with devices that can only be accessed from particular processors), the agents are location-transparent. The network of functions is driven by the values of the physical input attributes (*I*), which the underlying middleware may provide.

As with design classes, designing an agent is accomplished by assigning attributes to the five named compartments of the UML class representing the agent. Assigning attributes to the *provides output*, *requires output*, and *local* compartments of the agents records the run-time model's decomposition of the behavioral model. Implicit in assigning an attribute to one of these compartments is the assigning of the corresponding value function to the agent.

Two compartments—*provides output* and *provides input*—record the *provides interface* of the agent, representing the public interface it presents to others. It is intuitive to think of the middleware establishing a named write-only or read-only port for each attribute in the two compartments. For an attribute in the *provides output* compartment, the middleware establishes a read-only port from which others can obtain the value of the attribute. For an attribute in the *provides input* compartment, the middleware establishes a write-only port. The agent sets the attribute to a particular value when the write-only port receives the value, which will result in changes to the values of attributes whose functions depend upon the former attribute.

Two compartments—*requires input* and *requires output*—record the *requires interface* of the agent, representing what the agent requires, e.g., of the *provides interfaces* of other agents. Again, it is intuitive to think of the middleware establishing a named, write-only or read-only port for each attribute. The middleware establishes a write-only port for each attribute assigned to the *requires input* compartment. The middleware may provide physical inputs to an agent whose *requires input* compartment lists their names. Similarly, the middleware establishes a read-only port for each attribute assigned to the *requires output* compartment. The middleware may provide to appropriate physical devices values of physical outputs listed in an agent's *requires output* compartment.

3. Exercising the Sage Tool Set

The prototype Sage toolchain includes the Sage prototype tool set, sol2sal compiler, Salsa property checker [6], Sol compiler [4], and SINS middleware [5] which provides an execution environment for Sage agents. The prototype Sage tool set comprises a set of plug-ins to the Eclipse IDE (integrated development environment) and a set of external programs and scripts. The Eclipse Modeling Framework (EMF) [8] generated several of the plug-ins from the Sage metamodel captured in Ecore, the modeling language of EMF. The external programs and scripts, in addition to generating graphical views of the models, generate agents in the SOL language [4] from a Sage run-time model.

Of the several applications which have exercised the toolchain, the largest and most complex is the WCP (weapons control panel) [14]. Operators of a deployed US military platform use the WCP to monitor and prepare weapons for launch. The contractor-developed SRS, which was translated into a Sage behavioral model, comprises 258 variables—108 inputs, 90 outputs, and 60 internal variables—and 150 functions.

Sage was used to create an environmental model of the WCP, consisting of five environmental classes, one for each of three panels and two external mechanical assemblies. Several run-time models of WCP were developed using Sage (the Sage run-time model dictionary supports an arbitrary number of run-time models in an application model). One was a single, monolithic agent. Another was a six-agent model, consisting of one agent for each of the five objects described by the environmental model and an additional agent to compute shared values. The sol2sal compiler translated a SOL model of the WCP, which Sage generated, to the language of Salsa. Salsa checked the consistency and completeness [13] of the WCP, finding one non-deterministic function. Sage has generated SOL for both run-time models. After compilation by the SOL compiler both models have been deployed and executed on SINS middleware, and demonstrated using scenarios developed for a previous version of the WCP [14]. A SINS policy enforcement agent for the WAP_SAFE_1 safety property [14] was developed in Sage. The policy enforcement agent has been deployed, executed, and shown to detect violations of the safety property on the SINS middleware by both the single agent and six-agent versions of the WCP.

4. Conclusions

We have described Sage, a method and its model-driven tool set. We have built a prototype toolchain and have used it to construct, automatically check for consistency and completeness, and execute in the SINS environment, several example multi-agent systems, including the WCP, a subsystem on a deployed weapon platform. Deploying WCP in several configurations demonstrates reconfigurability.

That Sage agents can be automatically translated and executed demonstrates that Sage can precisely record behavior. However, Sage captures behavior at a relatively high level in that a concern of all MAS methods we've examined, the contents of messages and protocols for exchange of messages among agents, is not a concern of the Sage user when deploying agents in SINS. This may reflect the application domain we're addressing—reactive systems.

The four Sage models provide at least some of the documentation required of high assurance systems. That executable software is produced from the models suggests that agile development could produce high assurance software and the associated required documentation. Further, the nature of the four models supports agile development. The models can be produced in any order, allowing an agile development project to react quickly to customer needs. Partial and inconsistent models are executable facilitating rapid delivery to the customer for use and evaluation.

5. Related Work

Amor and colleagues [1] note that there are a variety of agent-oriented methodologies focusing on analysis and design, but "they do not completely resolve how to achieve the model derivation from the system design to a concrete implementation." Amor and colleagues want to be able to transition from an arbitrary MAS design methodology to an arbitrary agent platform. They propose an approach based on the OMG's Model Driven Architecture (MDA), which will transform a MAS design to Malaca, their "platform-neutral" agent architecture, and then transform the Malaca-based agents to the desired agent platform. They note that such transformations can not be automated when the design provides insufficient information. The Sage effort demonstrates "how to achieve the model derivation from the system design to a concrete implementation" for a particular methodology and agent platform. Transforming Sage models to a particular agent platform demonstrates that Sage models contain sufficient information for automation.

Cossentino and Potts [11] describe PASSI, an elaborate MAS design methodology and process. Use-case diagrams describe functional requirements for the MAS as a whole. Packaging the use-cases identifies agents and assigns functionality to them. The process concludes with the design of the methods of the classes implementing each agent "using classical approaches (like flowcharts, state diagrams and so on) or event semi-formal text descriptions" [11], code production, and deployment configuration. They propose to move beyond generation of class skeletons by introducing standard pieces of code into method bodies based on models from earlier phases. The Sage behavioral model captures MAS functionality more precisely than is possible with use-cases and, like PASSI, assigns it to agents. Sage supports an agile, opportunistic process that does not require redundant capturing of developer decisions, in contrast to PASSI's traditional waterfall process.

Subsequently, Chella, Cossentino, and colleagues developed Agile PASSI [9][10] because developers found that development with PASSI was not as fast and flexible as they would like. Agile PASSI focuses on production of code rather than documentation, simplifying the elaborate documentation required by PASSI. In addition, an add-in for the MetaEdit+ tool generates some documentation. Agile PASSI uses the Agent Factory tool to generate code skeletons from diagrams and fill in method bodies with reused code. Manual coding completes the implementation. Sage both provides documentation required by high assurance (which Agile PASSI eschews) and generates code. On the examples we've worked, the recording of behavior in the Sage behavioral model obviates the need for manual coding.

The UML-RSDS method and tools [16][17][18][19] support model-driven development for reactive systems by generating executable code from specifications. The tools translate to the B language a selected subset of UML, including behavior, inheritance, and associations among classes captured by constraints written in LOCA, an OCL-like based on the B language. Static invariants are checked in the resulting B. Tools can generate Java and C, and SMV for temporal property checking. The approach does not address decomposing software into agents or run-time components, assigning behavior to them, nor describing the interfaces among them.

Acknowledgements

Comments from the reviewers led to significant improvements in the paper. M. Li, B. Montrose, and A. Khashnobish implemented significant portions of Sage. N. Padh's efforts were crucial to the deployment of WCP on SINS. The author is grateful to S. Chincheck and M. Kang for their support of this work.

References

[1] M. Amor, L. Fuentes, and A. Vallecillo. "Bridging the Gap Between Agent-Oriented Design and Implementation Using MDA", *AOSE 2004*, LNCS 3382, pp. 93-108, Springer-Verlag 2005.
[2] K. Beck *et al*., "Manifesto for Agile Software Development", Agile Alliance, 2001, http://agilemanifesto.org/.
[3] C. Bernon, M. Cossentino, and J. Pavon. "Agent Oriented Software Engineering", *Knowledge Engineering Review* (in printing), www.pa.icar.cnr.it/~cossentino/publications.htm.
[4] R. Bharadwaj. "SOL: A Verifiable Synchronous Language for Reactive Systems", *Proc. Synchronous Languages, Applications, and Programming*, Electronic Notes in Computer Science, Elsevier, 2002.
[5] R. Bharadwaj, "Secure Middleware for Situation-Aware Naval C2 and Combat Systems", *Proc. Ninth International Workshop on Future Trends of Distributed Computing Systems*, May 2003, San Juan, Puerto Rico.
[6] R. Bharadwaj and S. Simms. "Salsa: Combining Constraint Solvers with BDDs for Automatic Invariant Checking", Lecture Notes in Computer Science, Volume 1785, Jan 2000.
[7] K. Britton, R. Parker, D. Parnas. "A Procedure for Designing Abstract Interfaces for Device Interface Modules", *Proc., 5th International Conf. on Software Eng.*, March 1981, pp.195-204.
[8] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework: A Developer's Guide*, Pearson Education, Boston, 2003.
[9] A. Chella, M. Cossentino, L. Sabatucci, and V. Seidita. "From PASSI to Agile PASSI: tailoring a design process to meet new needs", *Proc. of the IEEE/WIC/ACM International Conf. on Intelligent Agent Technology*, 2004.
[10] A. Chella, M. Cossentino, L. Sabatucci, and V. Seidita. "Agile PASSI: An Agile Process for Designing Agents", *International Journal of Computer Systems Science and Eng*., Special issue on "Software Eng. for Multi-Agent Systems", May 2006.
[11] M. Cossentino and C. Potts. "A CASE tool supported methodology for the design of multi-agent systems", *International Conference on Software Engineering Research and Practice*, June 2002, Las Vegas, USA.
[12] R. Guindon, "Designing the Design Process: Exploiting Opportunistic Thoughts", *Human-Computer Interaction*, Lawrence Erlbaum Associates, Inc., 1990, Vol. 5, pp. 305-344.
[13] C. Heitmeyer, R. Jeffords, and B. Labaw. "Automated Consistency of Requirements Specifications", *ACM Trans. Software Eng. and Methodology*, Vol. 5, No. 3, July 1996.
[14] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications", *IEEE Trans. on Software Eng*., Vol. 24, No. 11, November 1998.
[15] J. Kirby, "Rewriting Requirements for Design". *Proceedings, IASTED International Conference on Software Engineering and Applications (SEA) 2002*, November 4-6, 2002, Cambridge, MA, USA.
[16] K. Lano, K. Androutsopoulos, and D. Clark. "Structuring and Design of Reactive Systems Using RSDS and B", FASE 2000, LNCS 1783, p. 97-111, Springer-Verlag, 2000.
[17] K. Lano, D. Clark, and K. Androutsopoulos. "RSDS: A Subset of UML with Precise Semantics, December 13, 2001.
[18] K. Lano, D. Clark, and K. Androutsopoulos. "From Implicit Specifications to Explicit Designs in Reactive System Development", IFM 2002, LNCS 2335, pp. 49-68, Springer-Verlag 2002.
[19] K. Lano, D. Clark, and K. Androutsopoulos. "UML to B: Formal Verfication of Object-Oriented Models", IFM 2004, LNCS 2999, pp. 187-206, Springer-Verlag 2004.
[20] D.L. Parnas and P.C. Clements, "A Rational Design Process: How and Why to Fake It", *IEEE Trans. on Software Engineering*, IEEE Computer Society, February 1986, pp. 251-257.
[21] D.L. Parnas and J. Madey, "Functional Documentation for Computer Systems Engineering", *Science of Computer Programming*, Elsevier, October 1995, pp 41-61.
[22] D. Parnas, P. Clements, and D. Weiss. "The Modular Structure of Complex Systems", *IEEE Trans. on Software Engineering*, IEEE Computer Society, March 1985.
[23] "Principles behind the Agile Manifesto", Agile Alliance, http://agilemanifesto.org/principles.html.
[24] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, Addison-Wesley 1999.
[25] B. Selic, "The Pragmatics of Model-Driven Development", *IEEE Software*, IEEE Computer Society, September/ October 2003, pp. 19-25.